LabAssistant: beneficios del uso de un lenguaje reflexivo en un sistema adaptable por el usuario

Gustavo Wolfmann
LIDeS - Laboratorio de Investigación y Desarrollo de Software
Fac. Cs. Exactas Físicas y Naturales - Universidad Nacional de Córdoba
Córdoba - Argentina - Te/Fax +(54 351) 4334409 / 4334136
$gwolfmann@efn.uncor.edu$

Resumen

LabAssistant es un proyecto de desarrollo de software para los laboratorios de ensayos científicos destinado a sistematizar la información de los ensayos que realizan. Al ser un sistema de alcance general para todo el sector, se impone poder configurar la cantidad y calidad de los atributos de información a registrar por cada tipo de ensayo. A partir de un diseño con un número fijo de clases semidefinidas, se decidió implementar en smalltalk ya que posee capacidades reflexivas avanzadas y ausencia de tipado explícito. Estas propiedades facilitan el desarrollo y la mantenibilidad de la aplicación en comparación con una implementación hecha en un lenguaje que carezca de ellas, al posibilitar un código homogéneo respecto de los atributos predefinidos y los configurados y facilitar el testing de la aplicación.

1. Introducción

El proyecto LabAssistant surge a partir de una convocatoria de la Agencia Córdoba Ciencia destinado al desarrollo de proyectos de software. Tiene como finalidad la sistematización de la información de ensayos practicados por los laboratorios científicos. Es un proyecto de desarrollo de alcance general para todo el sector, por cuanto debe poseer capacidad de almacenar información de numerosos tipos de ensayos, tan variados como por ejemplo, los ensayos de resistencia de materiales, poder germinativo de semillas o propiedades eléctricas de artefactos de uso doméstico.

Existen laboratorios de gran envergadura que poseen sistemas adecuados a sus requerimientos, pero la gran mayoría no poseen sistemas que permitan sistematizar la información de los ensayos; más bien trabajan con planillas o pequeñas bases de datos sin una visión sistémica de la información. Sin embargo, esta última es de vital trascendencia si el laboratorio quiere certificar normas ISO, sobre todo en lo referido a la trazabilidad y resguardo de la información respaldatoria de las certificaciones que extienden. Por tanto, el objetivo del proyecto es desarrollar un sistema que permita cubrir esas necesidades.

Cada tipo de ensayo posee un protocolo particular, por el cual se deben cumplir el procedimiento, las etapas y las pautas de información prescriptas. Es un sistema inherentemente adaptable a cada usuario, donde se debe poder configurar para cada tipo de ensayo, la secuencia de etapas a desarrollar, y para cada una de estas, una serie abierta de atributos de información, tanto en su cantidad como al tipo de dato a contener.

2. Metodología de relevamiento y diseño

El relevamiento de los requerimientos se practicó en tres laboratorios de la Universidad Nacional de Córdoba, se siguió la metodología de entrevistas abiertas para recabar información de los tipos de ensayos practicados en cada uno de ellos y se analizo la documentación utilizada para la realización de los ensayos, tanto en formato físico como electrónico. Finalmente se volcó lo relevado en un documento de Léxico Extendido de Lenguaje (LEL) [HKOdPL97] y otro de Casos de Uso para cada laboratorio. Se resumieron los requerimientos de software de los tres laboratorios en un único documento de requerimiento de software, basandose en la metodología de Object Behaviour Analysis, de K. Rubin y A.Goldberg [RG92], con facilidades de trazabilidad de los requerimientos a los casos de uso respectivos.

A partir de los requerimientos se realizó el diseño conceptual de los componentes del sistema. Se definieron los entidades conceptuales intervinientes en el mismo y las relaciones entre ellos, lo cual puede verse en el diagrama conceptual de la figura 1. En él están expresadas todas las entidades que intervienen en el sistema sin detallar los atributos de información que cada una de ellas poseen.

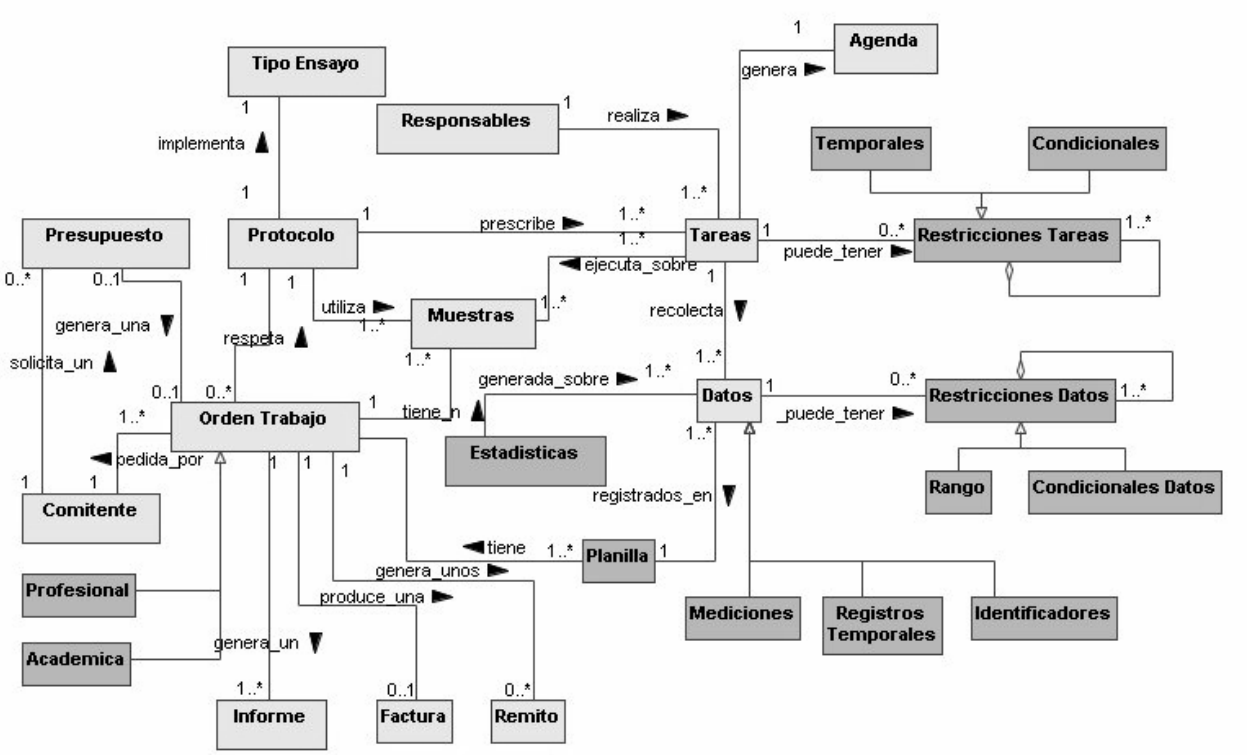

Figura 1: Diagrama Conceptual del dominio relevado

El sistema provee una clase para casi todas las entidades conceptuales existentes, de forma tal que existe una relación biunívoca entre entidades conceptuales y clases implementadas. En este sentido decimos que el sistema se configura a partir de clases semidefinidas: las clases del sistema están predefinidas, como así también los atributos que son comunes a todas las configuraciones del sistema, quedando a cargo del usuario final definir para cada clase, cuales son los atributos que son propios de sus requerimientos. Este diseño esta basado en una variación de Hruby [Hru05] respecto de construir una aplicación derivando clases a partir de una ontología, ya que este autor plantea el caso

de la generación de una familia de aplicaciones, no una aplicación configurable por el usuario como es nuestro caso. Otra diferencia con dicho autor es que él configura los requerimientos específicos de una aplicación por intermedio de aspectos, los cuales necesita tener perfectamente predefinidos al momento de ensamblar la aplicación. En nuestro caso, desconocemos cuales son dichos requerimientos específicos, ya que solo fueron relevados tres de todos los laboratorios posibles.

3. Decisiones de implementación

El tipo de sistema planteado podría haber sido implementado como una familia de sistemas, donde el equipo de desarrollo configura la aplicación específica a cada laboratorio, o bien con una estrategia de autogeneración del usuario fina, donde este sea el encargado de configurar los parámetros de generación de la aplicación, y además deba compilarla e instalarla en sus equipos. Ninguna de las dos opciones resultaron ser viables, ya que en el primer caso se plantearía una dependencia extrema del usuario con el equipo de desarrollo, y tampoco resulta viable la segunda, ya que el usuario final debería estar capacitado para poder generar su propia aplicación, cosa muy poco frecuente de hallar dentro del personal de un laboratorio. Es por esto que el sistema debe venir ya compilado y debe ser instalado y adaptado in-situ por cada usuario lo más sencillamente posible.

Por otro lado, la gran mayoría de estos laboratorios no poseen más que unas pocas computadoras conectadas en una red local, lo que resta sustento a soluciones del tipo *web-enabled* con un servidor dedicado y con motores independientes de base de datos, volcando la balanza hacia una aplicación del tipo *stand-alone*.

Las características del sistema imponen el uso de algún tipo de metaprogramación para poder llevar adelante la aplicación. P.Maes [Mae87] señala una distinción entre una meta-arquitectura y una arquitectura reflexiva: la primera da un acceso estático a la representación de un sistema computacional mientras que la segunda da un acceso dinámico, a lo cual Demers y Malenfant [DM95] agregan que los cambios que se pueden realizar en una meta-arquitectura son previos a la ejecución del programa, mientras que en una arquitectura reflexiva son en *run-time*. La *auto-inspección* es derivada de la primera y la *auto-adaptación* de la segunda. Solo algunos lenguajes de programación poseen la primera y muchos menos la segunda, pero ambas suelen aludirse como características reflexivas de un lenguaje.

Llevando estos conceptos reflexivos al nivel de los lenguajes orientados a objetos la *auto-inspección* hace referencia a la capacidad de un objeto de informar a otros cuales son los mensajes que es capaz de responder mientras la *auto-adaptación*, a más de lo anterior, permite modificar en *run-time* los mensajes que es capaz de responder. Es decir, se pueden agregar, eliminar y modificar los mensajes que una instancia de una clase responderá en tiempo de ejecución.

La decisión del lenguaje de implementación cayó finalmente sobre smalltalk (versión Visualworks), por sus características de ausencia de tipado explícito, reflexividad tanto en lo referido a la *auto-inspección* como a la *auto-adaptación*, capacidad de generar una aplicación *stand-alone* fácilmente instalable, disponibilidad de herramientas para el diseño de la interface con el usuario y para la persistencia de los datos, pero fundamentalmente gravitó sus características reflexivas.

La configuración que el sistema le permite definir al usuario final para adaptarlo a sus necesidades, es leída por un proceso de *start-up* del sistema que determina cuales son los atributos que serán necesarios agregar a cada clase semidefinida, y su tipo de información.

De esta forma, por cada atributo agregado por el usuario en una clase, se generan dos cambios en esta, a saber, agregarlo como variable de instancia de la clase y generar de sus respectivos métodos accesores. La *auto-adaptación* de smalltalk permite hacer esto en forma directa sobre una clase, de forma tal que luego de concluido este proceso de *start-up*, no existe ninguna diferencia en como referenciar los atributos predefinidos de los definidos en *run-time* por el sistema.

Pongamos un ejemplo: la clase Orden de Trabajo tiene atributos como número, fecha, comitente y otros más que son comunes a toda la familia de configuraciones, por lo que el sistema los trae ya predefinidos, ahorrando tiempo al usuario final. Sin embargo, puede ser que el atributo "cantidad de muestras" sea necesario en algunos casos, por cuanto, quien lo requiera, deberá agregarlo en la configuración de su sistema. Supongamos que llamamos internamente a este atributo como *num_muest*. Luego del proceso de *start-up*, cualquier objeto instancia de la clase orden de trabajo tendrá un método *num_muest* y *num_muest:* que sean accesores a la variable de instancia *num_muest*, de manera similar a los métodos predefinidos como *num_orden*, *fecha*, etc.

4. Beneficios obtenidos

La capacidad de *auto-adaptación* de smalltalk nos permite tratar de idéntica forma los atributos predefinidos como los configurados por el usuario. Esto resulta en una ganancia a la hora de la codificación ya que no hay condiciones especiales según sea el origen del atributo por lo que el tratamiento de la persistencia, la generación de formularios y los reportes, es codificado de forma homogénea respecto del origen de los atributos. La ausencia de tipado explícito y la *auto-inspección* es particularmente beneficiosa en la generación de reportes: basta con consultar cuales son los atributos de tipo numérico existentes para una clase dada, y luego generar reportes y estadísticas utilizando estos atributos como mensajes a cada instancia.

Implementar este sistema en un lenguaje con tipado explícito y sin *auto-adaptación* hubiera generado una solución con un diseño complejo de clases; además los atributos configurables por el usuario se deberían implementar en una colección de éstos, y sus métodos accesores no existirían, más bien serían reemplazados por un método accesor general que tomaría como parámetro el nombre del atributo deseado, lo que generaría una diferenciación en el tratamiento de los atributos predefinidos respecto de los definidos por el usuario, obteniéndose un código "*sucio*". La presencia del tipado explícito provocaría la necesidad de *castings* en los atributos numéricos a la hora de las estadísticas en los reportes. Ambas consecuencias son particularmente peligrosas en el testing y *runtime* del sistema: en razón de que los atributos definibles no actuan como atributos directos de una instancia en particular, sino que son manejados indirectamente, un error de codificación en un *casting*, o en un nombre de un atributo en el código de persistencia, provocarían errores muy difíciles de detectar. Poco sentido tiene además utilizar *castings* en lenguajes con tipado explícito ya que se burla el principal argumento de este tipo de lenguajes: la inexistencia de la posibilidad de invocar metodos no comprendidos por el objeto.

5. Conclusiones

En un sistema de las características como el planteado, es altamente recomendable codificar la aplicación con un lenguaje sin tipado explícito y con características de refle-

xividad avanzadas. Un punto en contra de esta elección suele ser la falta de capacitación de los programadores en lenguajes de estas características. Lamentablemente los fondos disponibles para este proyecto no han sido suficientes para poder desarrollar dos implementaciones en paralelo, usando ambos tipos de lenguajes y así poder hacer un estudio comparativo de las soluciones con observaciones reales de los tiempos e inconvenientes hallados. Pero acumulando la experiencia actual a las previas en desarrollos de sistemas, estimamos que el costo de la capacitación en smalltalk se paga en un ahorro en tiempos y costos de programación, facilidades de mantenibilidad del sistema y seguridad de que en el código no surgirán sorpresas ocultas causadas por los *castings*.

Este proyecto está soportado financieramente por al Agencia Córdoba Ciencia, y estará disponible bajo la modalidad de software libre para la comunidad. El grado de avance del mismo al momento de este escrito es el desarrollo de un prototipo específico para uno de los laboratorios relevados, almacenando su configuración en archivos XML. Puede consultarse en http://lides.efn.uncor.edu/labassistant.

Referencias

[DM95] F. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, August 1995.

[HKOdPL97] G. Hadad, G. Kaplan, A. Oliveros, and J. Sampaio do Prado Leite. Construcción de escenarios a partir del léxico extendido del lenguaje. In *Proceedings SoST97- 26 JAIIO,Sociedad Argentina de Informática y Comunicaciones*, Abril 1997.

[Hru05] Pavel Hruby. Ontology-based domain-driven design. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, October 2005.

[Mae87] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[RG92] Kenneth Rubin and Adele Goldberg. Object behavior analysis. *Communications of the ACM*, September 1992, Vol 35, No 9 1992.